



THESIS PAPER

**DESIGNING A SEGMENTED, INDEXED DATA WAREHOUSE AND
PERFORMANCE ANALYSIS AND IMPROVEMENT OF ASSOCIATION RULES
MINING WITH THE APRIORI ALGORITHM ON A VIRTUAL SUPERMARKET
DATABASE**

by

SHAHTAB MAHMUD (07101012)

SUPERVISOR : DR. MUMIT KHAN

Abstract

The apriori algorithm is a popular algorithm for association rules mining. This study on the apriori algorithm implements it on a segmented and indexed data warehouse and analyses the performance of it and comprehends possible approaches to the optimization of the algorithm.

The study concludes that the frequent pattern tree generation method can be optimized by segmentation, but the improvement is not substantial.

Table of Contents

1. Introduction	6
a. Acknowledgment	6
b. Literature Review	7
c. Proposition	10
2. The Data Warehouse	11
a. Schema of the data Warehouse	16
3. Implemetation	17
a. Data Pre Processing	17
b. Indexing	18
c. The Apriori Algorithm	19
d. Frequent Pattern Growth	20
e. Implementation	21
4. Performance Analysis and Optimization	24
a. Performance analysis	24
b. Performance optimization	25
c. Findings	26
5. List of references	29
6. Source codes	30

Table of Figures

1. Data Warehouse	11
2. Enterprise warehouse with data store	13
3. Warehouse schema	15
4. Warehouse Schema	16
5. B-Tree	18
6. Hierarchical data model	18
7. Tree generation graph	25
8. Pruning non indexed tables	26
9. Relative performance chart	27

Acknowledgement

Foremost, I would like to express my sincere gratitude to my supervisor Dr. Mumit Khan for the continuous support of my undergraduate study. His patience, guidance, immense knowledge and enthusiasm guided through my research. I could not have hoped for a better supervisor than Dr. Khan.

Besides my supervisor, I would like to express my gratitude to all member of the thesis committee, specially Dr. Khalilur Rhaman. My peers especially Syed Sabbir Ahmed, Tahniat Ashraf Priyam and Brian D'Rozario deserve much credit for bearing with me.

Last but not the least, I thank my wonderful family and God for keeping my endowed with their blessings.

Literature Review

Database applications can be broadly classified into transaction processing and decision support systems. Transaction processing systems are systems that record information about transactions such as product sales information or course registration and grades for universities.

Transaction processing systems are widely used today and companies have amassed huge amount of data. Decision support systems use the detailed information stored in transaction processing systems to generate high level information to make a variety of decisions. DSSs help managers decide what products to stock, what products to manufacture or even which products should be placed adjacently as these products are often sold in a bundle.

The term *Data Mining* refers the process of semi automatically analyzing large databases to find useful patterns. Similar to Machine Learning or Statistical Analysis, Data Mining attempts to discover rules and patterns from data. But Data Mining deals with large volumes of data. The knowledge discovered through mining are generally used for predictions and associations.[14]

One of the major interests of data mining is finding descriptive patterns like clusters or more arelevantly for this research, finding associations. The problem of association rule mining is defined as:

Let $I = \{i_1, i_2, \dots, i_n\}$ be a set of n binary attributes called items.
Let $D = \{t_1, t_2, \dots, t_m\}$ be a set of transactions called the database. Each transaction in D has a unique transaction ID and contains a subset of the items in I . A rule is defined as an implication of the form $X \Rightarrow Y$ where $X, Y \subseteq I$ and $X \cap Y = \emptyset$. The sets of items (for short itemsets) X and Y are called antecedent (left-hand-side or LHS) and consequent (right-hand-side or RHS) of the rule respectively.[8]

Association rule mining finds out 'interesting' association rules that satisfy the predefined minimum support and confidence from a given database. The problem is usually decomposed into two subproblems. One is to find those itemsets whose occurrences exceed a predefined threshold in the database; those itemsets are called frequent or large itemsets. The second problem is to generate association rules from those large itemsets with the constraints of minimal confidence.

The first sub-problem can be further divided into two sub-problems: candidate large itemsets generation process and frequent itemsets generation process. We call those itemsets whose support exceed the support threshold as large or frequent item- sets, those itemsets that are expected or have the hope to be large or frequent are called candidate itemsets. In many cases, the algorithms generate an extremely large number of association rules, often in thousands or even millions. Further, the association rules are sometimes very large. It is nearly impossible for the end users to comprehend or validate such large number of complex association rules, thereby limiting the usefulness of the data mining results. Several strategies have been proposed to reduce the number of association rules, such as generating only "interesting" rules, generating only 'nonredundant' rules, or generating only those rules satisfying certain other criteria such as coverage, leverage, lift or strength.[2]

Now, there are many methods used for Association Rules Mining; collaborative association rule generator, the AIS algorithm to name a few. The most widely used algorithm for association rules mining is the apriori algorithm.

Apriori is more efficient during the candidate generation process. Apriori uses pruning techniques to avoid measuring certain itemsets, while guaranteeing completeness. These are the itemsets that the algorithm can prove will not turn out to be large. [2]

However there are two bottlenecks of the Apriori algorithm.

- the complex candidate generation process that uses most of the time, space and memory.
- Another bottleneck is the multiple scan of the database.

The computational cost of the apriori algorithm and association rules mining on a whole can be reduced in four ways: [2]

- by reducing the number of passes over the database
- by sampling the database
- by adding extra constraints on the structure of patterns
- through parallelization

The initial purpose of this research was to reduce the computational complexity of the apriori algorithm by –

- Segmenting the Database
- Sub-structuring the database to induce hierarchies
- Indexing the elements
- At the same time enforce the database to be ordered at all times so that sequences and sub sequences can be identified
- Maintaining an access table that maps different segments of the database so that memory access can be easier.

Usually, Databases those are mined for association rules, are sequential in nature. Stock prices, money exchange rates, temperature data, product sales data, and company growth rates are the typical examples of sequence databases. Sequence or sequential databases are characterized by being ordered in design.[4]

Now, association rules mining or similarity searches intend on finding recurrent subsequences in the data that are stored in the database, having segments in the database increases the possible ways to find these subsequences.

As [1] and [4] explains, sequential data can be discretized by Fourier analysis and it is possible to find similar recurrent subsequences using Euclidean (average distance between two points, $a.a + b.b = c.c$) distance or Time Warping Distance (the smallest distance between two sequences transformed by time warping i.e. the sequences don't have to occur in the same time frame.) as similarity measures as long as:

- An ordered (must have a monotonically changing pattern) list of piece-wise segments from a sequence of data can be obtained.
- The data is indexed

However, initially, the idea was not to segment the data in the above mentioned format. Rather, it was to segment the data 'levelwise'. The database would have a hierarchical tree structure. Branch nodes would be categories and leaf nodes would be items in a category. The idea was, depending in associations

between levels, many of the branches would be pruned hence the algorithm will not have to scan the whole database multiple times.

This very idea is explained and vilified in. [5]

Apriori itself is quick as it prunes many of the candidates as k comes closer and gets smaller and smaller with time. The advent of the Frequent Pattern Tree required only two passes of the transaction database. So, it is fast and optimized[2]. Improving on this performance would be fast. A counting based algorithm may be possible but as it may decrease the number of database passes, it will increase other computations and consequently perform fairly similarly.

Now, the idea is that databases need to be designed in such a manner that the data in it can be easily mined and results of mining can be easily validated.

Proposition

This research will

- design a Data Warehouse for a supermarket,
- a virtual sample of this database will be mined to estimate performance using
 - Apriori Algorithm
 - Levelwise apriori algorithm
 - Subsequence searching
- The performances of these different approaches will be compared

The Data Warehouse

For analytical purposes, many supermarkets are maintaining data warehouses. Since supermarket databases have a healthy amount of transactions and analysis, architectural designs of most databases can be derived from the data warehouse of a supermarket.

A **Data Warehouse** is a:

- subject-oriented
- integrated
- time-variant
- non-volatile

collection of data in support of management decisions. [6]

It is simply a database for reporting and analysis.

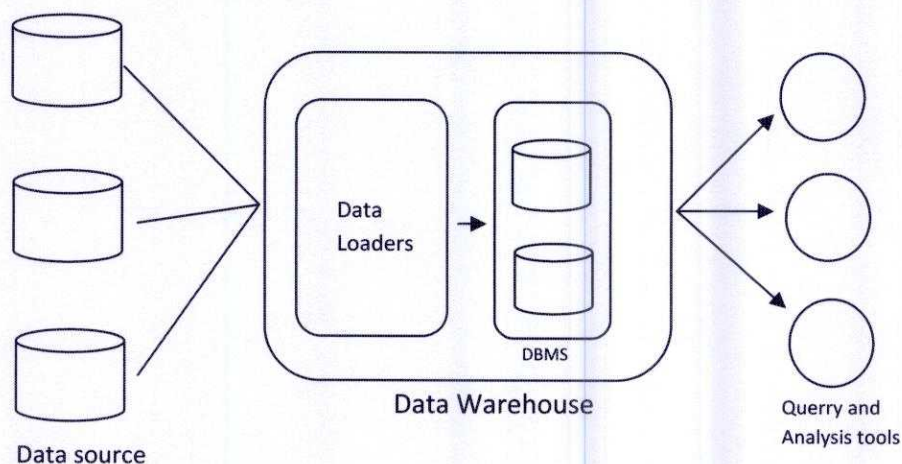
The data warehouse is oriented to the major *subject* areas of the corporation that have been defined in the data model. Examples of subject areas are: customer, product, activity, policy, claim, account. The major subject areas end up being physically implemented as a series of related tables in the data warehouse.

The second salient characteristic of the data warehouse is that it is *integrated*. This is the most important aspect of a data warehouse. Generally, there is no application consistency in encoding, naming conventions, physical attributes, measurements of attributes, key structure and physical characteristics of the data. Each application has been most likely been designed independently. As data is entered into the data warehouse, inconsistencies of the application level are undone.

The third salient characteristic of the data warehouse is that it is *time-variant*. A 5 to 10 year time horizon of data is normal for the data warehouse. Data Warehouse data is a sophisticated series of snapshots taken at one moment in time and the key structure always contains some time element.

The last important characteristic of the data warehouse is that it is *nonvolatile*. Unlike operational data warehouse data is loaded en masse and is then accessed. Update of the data does not occur in the data warehouse environment. [6]

The following figure is a simple model of a data warehouse described in [14], page 737.



A data warehouse can be of a source driven architecture if the sources are continually or periodically transmitting information to the warehouse.

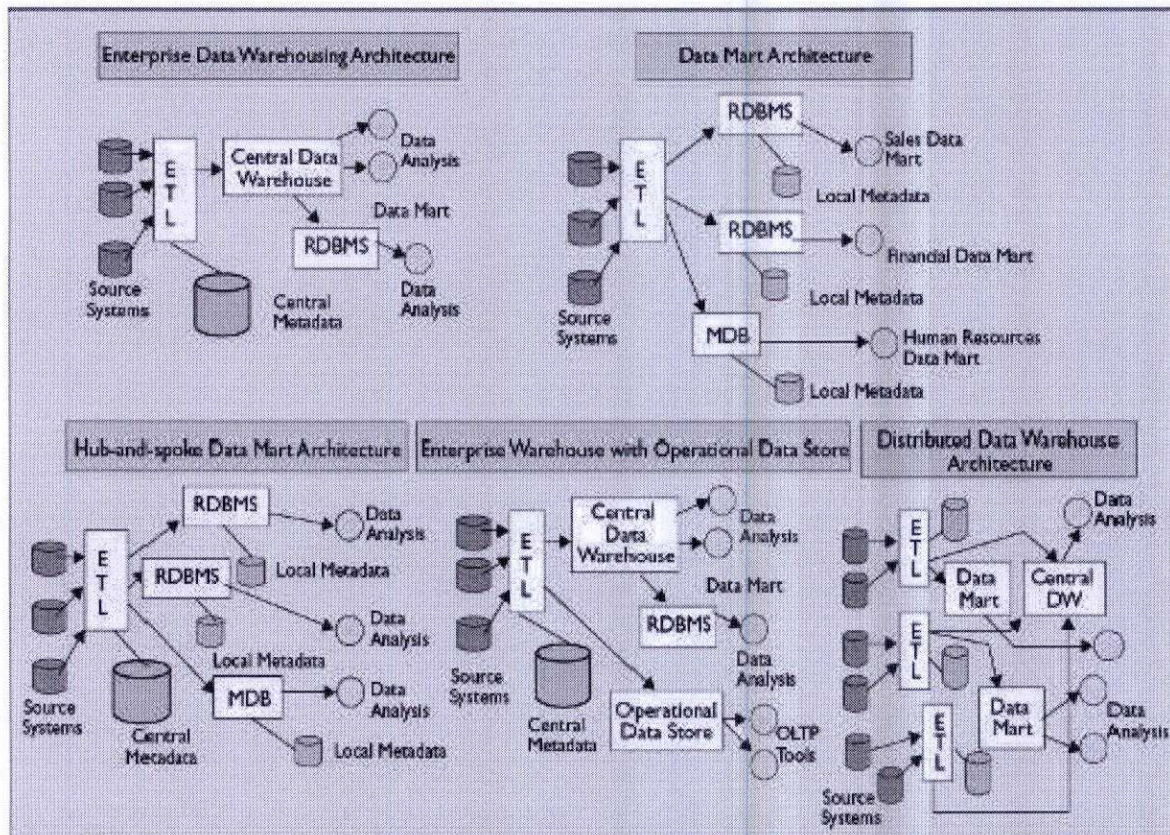
It can also be destination driven if the warehouse periodically sends requests to the data sources(s) for information.

Data sources are individual databases themselves and may or may not have different schemas and data models compared to the data warehouse. Part of a task of a warehouse is to integrate the schema and convert the data before they are stored. (data integration, Data Mining, ch 18 [14])

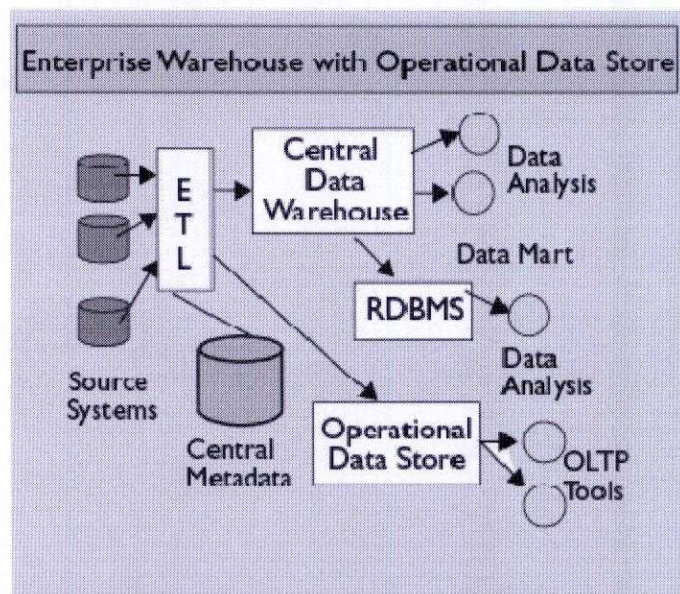
The data warehouse must also pre-process the data so that noise and errors can be efficiently mined. This process is called Data Transformation and cleansing.

After the warehouse has processed the data, it mines and then propagates the updates and summarizes the results. [14]

Warehouse Architecture for a Supermarket Chain



Depending on the size and the scale of the organization, the data warehouse can follow any of the structures. If need be, a data warehouse can be distributed as shown below.



Warehouse Schema

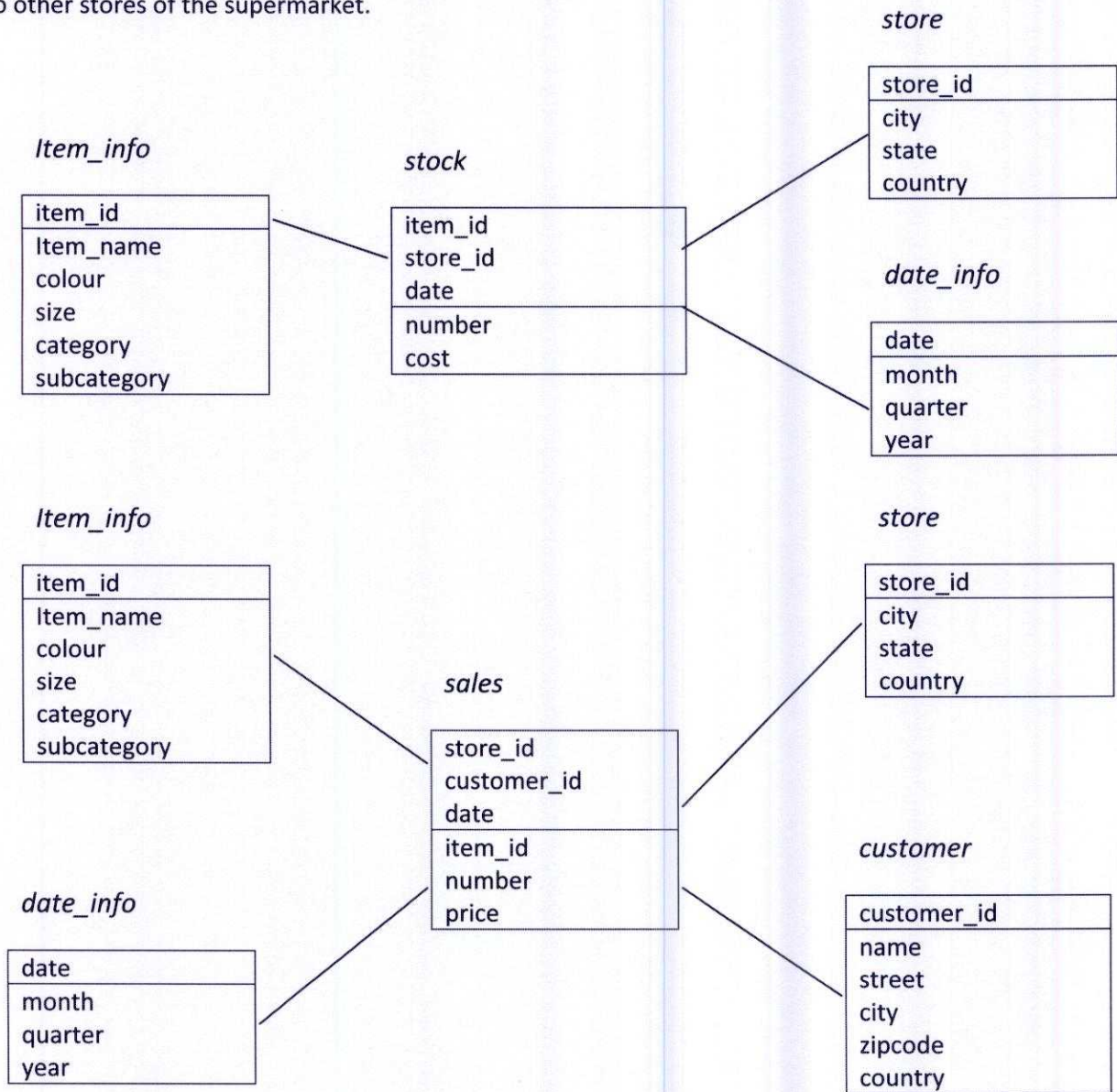
Data Warehouses are databases specifically built for analysis. So, they are optimized for OLAP (online analysis processing). The data are usually multidimensional, having *Dimension* attributes and *Measure* attributes. [14]

Tables containing multidimensional data are called *Fact Tables* and are usually very large.

Warehouse Schema for a Supermarket Chain

This is a star schema for a supermarket chain. To minimize storage requirements, dimension attributes are foreign keys to other tables.

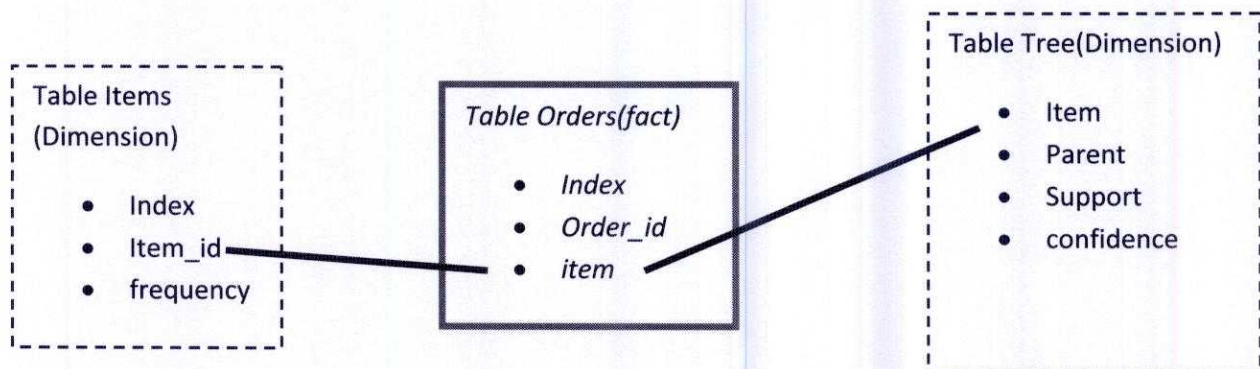
This is a warehouse with two fact tables. Stock and Sales. Real life warehouses are stores that simply sell to other stores of the supermarket.



Schema of the Project Warehouse

Since the objective of this project is to evaluate the apriori algorithm only,

- the schema only requires the table of orders as the fact table.
- The dimension tables are
 - The table of items and their frequencies
 - The frequent pattern tree table



Data Pre-processing (Transformation and Cleansing)

For subsequence searches, preconditions are that the data need to be

- Discrete
- Monotonous
- Storage and sales data are by default discrete. So, Fourier transformation for subsequence searches will not be necessary.
- Storage is not a type of data that the research intends to mine.
- For the sales data to be strictly monotonous, the list of items that are ordered, need to be sorted in ascending or descending order by the candidate keys. Candidate keys can be the categories or the item numbers themselves depending on the problem at hand.
- As data insertions would be coming from machines (barcode readers), checking spellings will not be necessary.
- The dataset needs to be uniform; hence, all orders that will be classified will have a fixed size of 5.

Indexing

Item_id's are densely indexed. However, Categories are intended to be used as sparse indices and will be fundamental in improving the performance of any classifier. Apriori is, of course, a classifier.

Most databases use B+ or B trees as data structures.

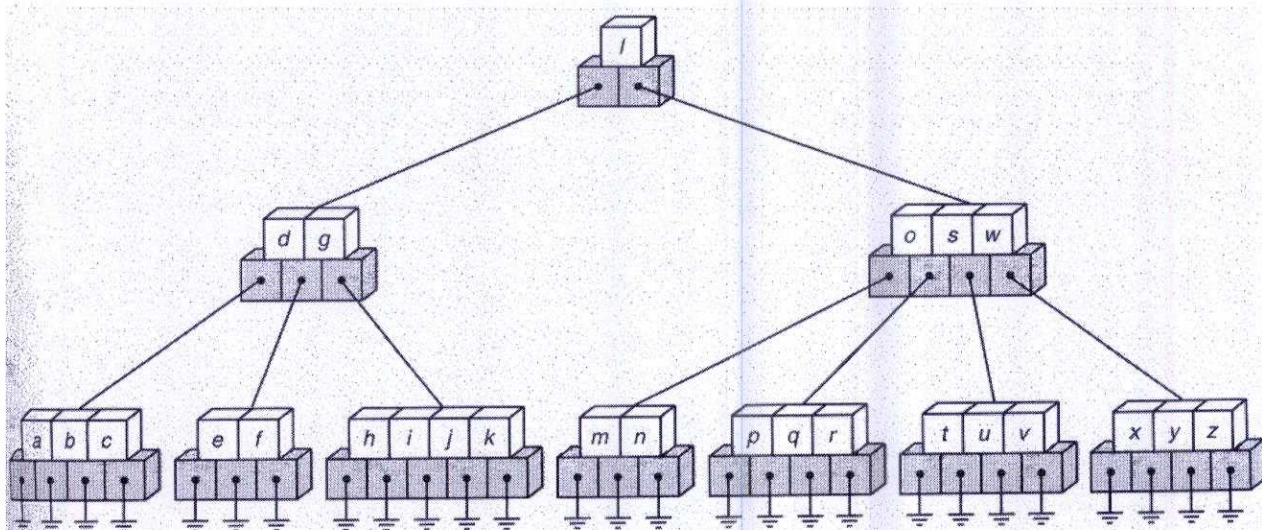
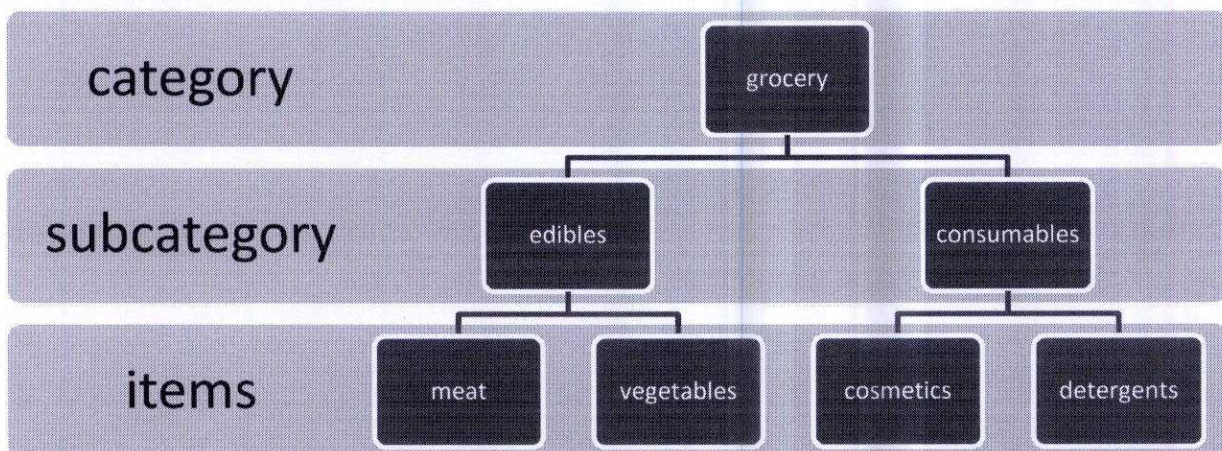


Figure 11.9. A B-tree

However, B trees ensure that the data are evenly distributed and that they are easily retrievable. The apriori algorithm prunes the infrequent sub trees. Hence, the larger amounts of data a pruned subtree will have, the better apriori will perform. This paper proposes a category based hierarchial data structure that resembles the following illustrations.



Since this is an indexed data warehouse, the hierarchies are treated as index levels. An item having the id 345 means, the category no. is 3, subcategory no. is 4 and the item no. is 5.

The Apriori Algorithm

Key Concepts :

- Frequent Itemsets: The sets of item which has minimum support (denoted by L for i th-Itemset).
- Apriori Property: Any subset of frequent itemset must be frequent.

Find the *frequent itemsets*: the sets of items that have minimum support

–A subset of a frequent itemset must also be a frequent itemset

•i.e., if $\{AB\}$ is a frequent itemset, both $\{A\}$ and $\{B\}$ should be a frequent itemset

–Iteratively find frequent itemsets with cardinality from 1 to k (k -itemset)

•Use the frequent itemsets to generate association rules.

•Join Step: C_k is generated by joining L_{k-1} with itself

•Prune Step: Any $(k-1)$ -itemset that is not frequent cannot be a subset of a frequent k -itemset

•Pseudo-code:

C_k : Candidate itemset of size k

L_k : frequent itemset of size k

$L_1 = \{\text{frequent items}\};$

for ($k = 1; L_k \neq \emptyset; k++$) **do begin**

$C_{k+1} =$ candidates generated from L_k ;

for each transaction t in database **do**

 increment the count of all candidates in C_{k+1} that are contained in t

$L_{k+1} =$ candidates in C_{k+1} with min_support

end

return $\bigcup_k L_k$;

Frequent Pattern Growth

- Compress a large database into a compact, Frequent-Pattern tree (FP-tree) structure
 - highly condensed, but complete for frequent pattern mining
 - avoid costly database scans
- Develop an efficient, FP-tree-based frequent pattern mining method
 - A divide-and-conquer methodology: decompose mining tasks into smaller ones
 - Avoid candidate generation: sub-database test
- First, create the root of the tree, labeled with “null”.
- Scan the database D a second time. (First time we scanned it to create 1-itemset and then L).
- The items in each transaction are processed in L order (i.e. sorted order).
- A branch is created for each transaction with items having their support count separated by colon.
- Whenever the same node is encountered in another transaction, we just increment the support count of the common node or Prefix.
- To facilitate tree traversal, an item header table is built so that each item points to its occurrences in the tree via a chain of node-links.
- Now, The problem of mining frequent patterns in database is transformed to that of mining the FP-Tree.

Implementation

The initial Idea was to program the whole algorithm using Java. Java hashtables and linked lists were perfect data structures for a simple implementation of the algorithm.

But handling large amounts of data became a potential problem. Moreover, association rules are mined on time variant databases hence, not using a relational data base made little sense.

The focus then shifted towards using Oracle as a relational database and using Java as the programming language. Oracle is industry standard and difficult to handle. The time and attention needed to configure Oracle and Java Database Connectivity (JDBC) drivers made it infeasible. (the JDBC manual for Oracle is 138 pages and the administration guide is 1068 pages).

So, PHP and MySQL on a XAMPP server has been used to implement the project.

The Data Dictionary view of the Tables on the database is:

item_freq

Column	Type	Null	Default	Comments	MIME
item_id	int(11)	No			
frequency	int(11)	No			

joined

Column	Type	Null	Default	Comments	MIME
<u>key</u>	int(11)	No			
order_id	int(11)	No			
item	int(11)	No			
frequency	int(11)	No			

orders

Column	Type	Null	Default	Comments	MIME
<u>key</u>	int(11)	No			
order_id	int(11)	No			
item	int(11)	No			

tree

Column	Type	Null	Default	Comments	MIME
<u>index</u>	int(11)	No			
id	int(11)	No			
count	int(11)	No			
sup	float	No			
conf	float	No			
parentid	int(11)	No			
nleft	int(11)	No			
nright	int(11)	No			
nlevel	int(11)	No			

The source codes of the various files used for this project are included at the end of the report.

Performance Analysis

The algorithm works in the following steps:

- The first pass through the database calculates the frequency of all the items that have been ordered.
- If the database contains n items, and the cost of calculation of frequency is constant, the complexity of the algorithm is nc .
- The second pass retrieves the number of items in a single order which is an inner join of the fact and dimension tables, in an rdbms is constant. Which is again, nc .
- On the second pass, the frequent pattern tree is generated, and the entry to the tree is either inserted or updated, which is constant. For n number of items, the complexity is nc .
- After the tree is generated, it is then pruned, which is a select statement where minimum support constraint is applied.
- For n items, the complexity is nc .
- The complexity of the whole algorithm is then $nc + nc + nc + nc + nc$.
- The Algorithm appears to be linear.

Performance Optimization

The basic idea of using segmented data for the frequent pattern generation was similar to Game Tree Pruning techniques used in Artificial Intelligence. If a decision is excluded from a scenario, then all its descendants too, are exempted from the scenario.

The apriori property denotes the all subsets of the frequent set are frequent and all subsets of infrequent sets are infrequent too. Using this property, we could assume that if a category of items were infrequent, the whole segment would be infrequent as well. This assumption is not true for all cases. But a classifier is not an empirical analysis, it's probabilistic. So, pruning the tree using this logic might help further optimise a widely used algorithm for association rules mining.

The tree generation will always include all items in a database. So, the only step that a segmented dataset can optimize is the pruning algorithm.

Since the tree implemented does not follow a node structure, the walk through the tree does not start from the root node and end at the leaves. The tree implemented is on a database table and it takes one select statement with a group by clause to generate the frequent set. This is equal to one pass over the tree table.

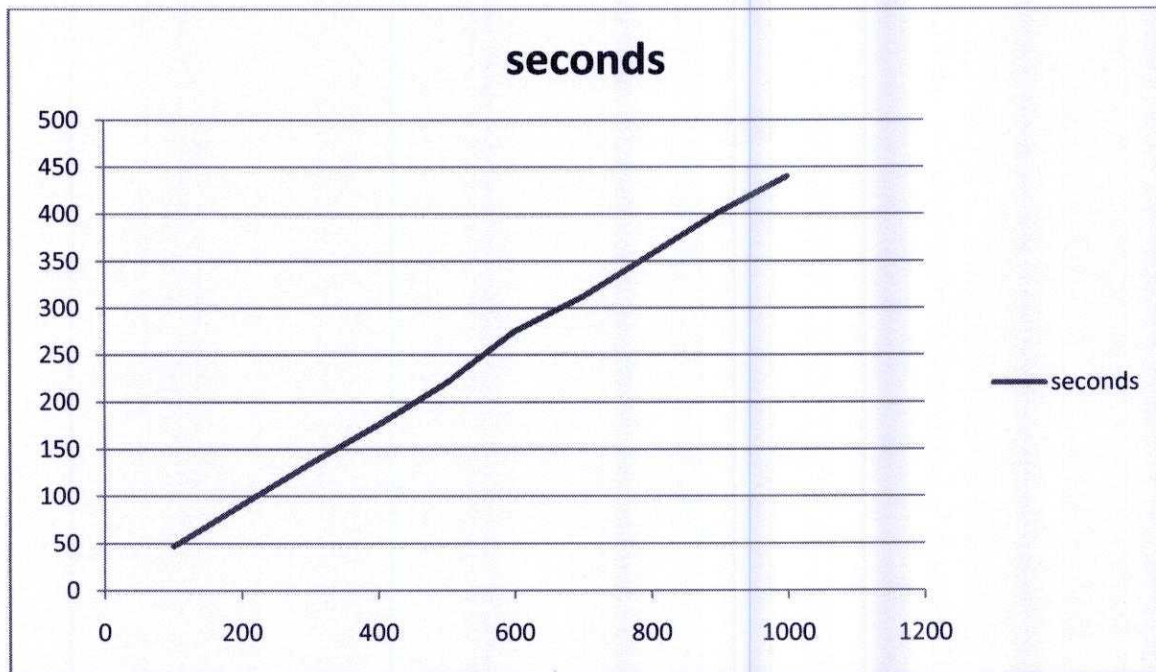
The study then divided the dataset into categories and subcategories and pruned the tree on the basis of the segmentation and compared the results.

Further, indexes were taken on the category and the subcategory markers and the improvement in performance was also accounted.

Findings

Tree generation:

Originally, the tree generation algorithm was thought to be a quadratic algorithm, but the actual performance ended up being linear because the passes over the database consisted of sql queries that perform in constant time. The completion time versus the number of values inserted graph shows the linearity of the tree generation.



Actual values:

100	47
200	91
300	135
400	176
500	220
600	275
700	312
800	357
900	402
1000	440

The study used a maximum of 15000 items in the orders, the process completed in approximately one hour and 30 minutes.

The prune step:

Non Indexed Tree Table

The prune step filters the tree table and returns the set of nodes that satisfy the constraints imposed on the basis of support and confidence.

The study used two tree tables. The first is not indexed. Only the primary key acts as index for this table. When pruned, the process is handled internally by the RDBMS and completes in microseconds.

Regardless of the no of nodes present on the tree, the prune step completes in constant time. The study pruned tables that contained 100, 500, 1000, 5000. 15000 items and it also took a dummy table that had 1000000 randomized values. All of these tables pruned in the same time. The performance can be explained with the B tree implementation that MySQL implements.

When pruned on all items, the step concludes in .0026 - .0029 seconds. However, when pruned on subcategory counts, the tables prune in .0012 - .0015 seconds.

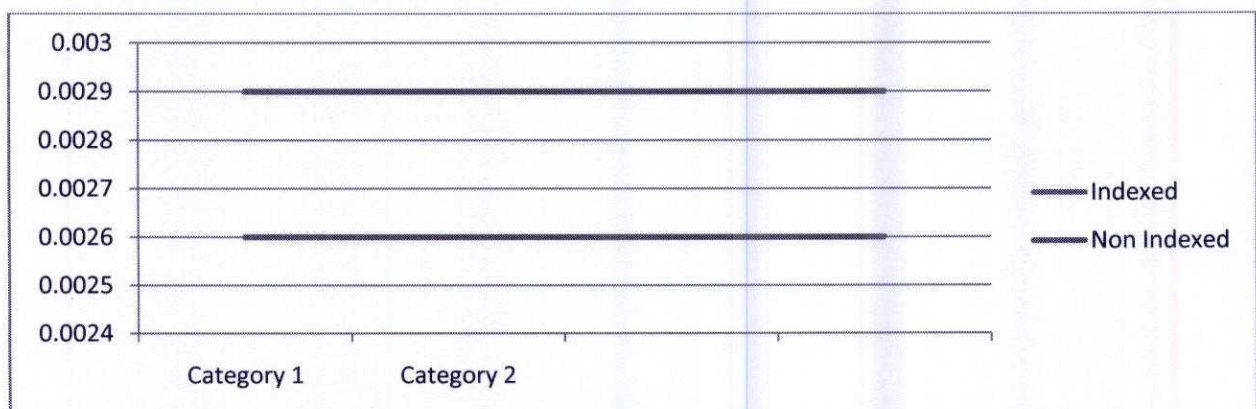
The finding corresponds to the assumption that using segmented datasets will yield faster completion. However, when the tree generation step takes hours to complete while data size is no more that thousands, it is an irrelevant improvement.

When the tree is pruned with one more segment in the dataset, the process concludes in .0008 - .0009 seconds.

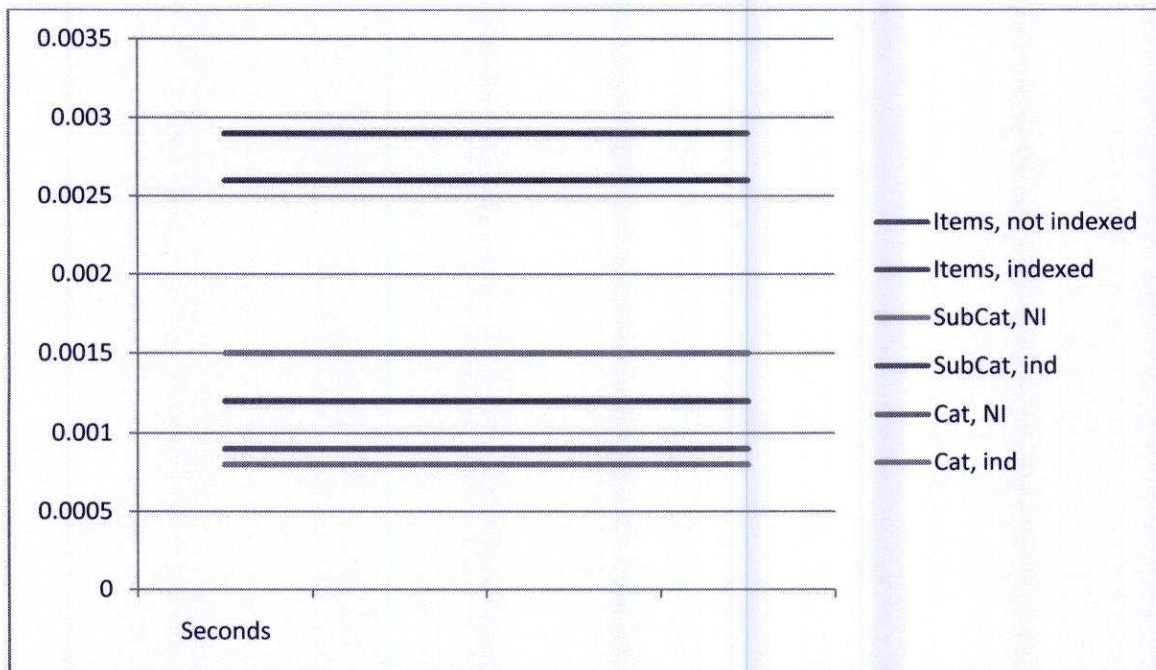
In theory, every segment within the dataset yields an improvement corresponding to the branching factor of the tree. In a top down Category – SubCategory – Item segmentation, there is a respective 66% (branching factor 1/3) and 50% (branching factor 1/2) improvement to the prune step itself.

Indexing the tables improve the performance of the algorithm too. The step always concludes .0003 seconds earlier. If the study had not used integer only indexes, the improvement may have been more profound.

Pruning non indexed tables using item counts vs indexed tables



Relative Performance chart:



List of References

1. Efficient Similarity Search In Sequence Databases
Rakesh Agarwal et al
2. Association rule mining: a recent overview
Sotiris Kotsiantis, Dimitris Kanellopoulos
3. A new adaptive support algorithm for association rules mining
Carolina Ruiz, Weiyang Lin, Sergio Alvarez
4. Segment based approach for subsequence searches in sequence databases
Sanghyun Park, S-W Kim and W W Chu
5. Mining Multiple-Level Association Rules in Large Databases
Jinway Han, Yongjian Fu
6. Data Warehousing: A Perspective
Hemant Kipekar
7. Data Warehousing and Decision Support
Torben Bach Pedersen
8. Mining Association Rules between Sets of Items in Large Databases
Agarwal et al
9. Database Architecture Optimized for the new Bottleneck: Memory Access
Peter Bonez
10. Architecture of a Database System
Joseph M. Hellerstein, Michael Stonebraker and James Hamilton
11. Data architecture for fetch-intensive database applications - Google Patents
12. Object Oriented Query Optimization
Gery Mitchell, Stanley Zdonik, Umeshwar Dayal
13. A Data Warehouse Architecture for Clinical Data Warehousing
Tony R. Sahama and Peter R. Croll
14. Database System Concepts
Abraham Silberschatz, Henry F. Korth, S. Sudarshan

Source codes

Config.php

```
<?php

error_reporting(E_ALL);
$messages=array();

include_once("functions.php");

$dbhost="localhost";
$dbuser="shahtab";
$dbpass="shahtab";
$dbname="warehouse";

connectToDB();
?>
```


Index.php

```
<?php
session_start();
include_once("config.php");

if(isset($_POST["submit"])) {

    set_time_limit(0);

    for ($o=1;$o<=1000;$o++){
        for ($i=1;$i<=5;$i++){

            $it_1 = mt_rand(1,5);
            $it_2 = (mt_rand(1,5) * 10);
            $it_3 = (mt_rand(1,5) * 100);
            $item= $it_1 + $it_2 + $it_3;

            process_order_table($o,$item);
        }

    }

}

if(isset($_POST["frequency"])) {

    set_time_limit(0);

    count_item_frequency();

}

if(isset($_POST["list"])) {
```

```
set_time_limit(0);
```

```
generate_tree();
```

```
}
```

```
if(isset($_POST["print"])) {
```

```
set_time_limit(0);
```

```
printab();
```

```
}
```

```
{  
?>
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"  
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">  
<html xmlns="http://www.w3.org/1999/xhtml" xml:Lang="en">  
<head>
```

```
<meta http-equiv="content-type" content="text/html; charset=iso-8859-1" />  
<meta name="description" content="your description goes here" />  
<meta name="keywords" content="your,keywords,goes,here" />  
<meta name="author" content="Your Name / Original design: Andreas Viklund -  
http://andreasviklund.com/" />  
<link rel="stylesheet" type="text/css" href="andreas02.css" media="screen,projection"  
title="andreas02 (screen)" />  
<link rel="stylesheet" type="text/css" href="print.css" media="print" />
```



```
<link rel="icon" type="image/x-icon" href="./img/favicon.jpg" />
<title>WAREHOUSE</title>
```

```
</head>
```

```
<body>
```

```
<form action="index.php" method="POST">
```

```
<center>
```

```
1.<input name="submit" type="submit" value="GENERATE ORDER TABLE"/><br/>
2.<input name="frequency" type="submit" value="COUNT ITEM FREQUENCY"/><br/>
3.<input name="list" type="submit" value="generate tree"/><br/>
4.<input name="print" type="submit" value="print"/><br/>
</center>
```

```
<br/><br/>
```

```
<br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/>
```

```
<h3></h3>
```

```
<p></p>
```

```
s
```

```
<p></p>
```

```
</form>
```

```
</body>
```

```
<?php
```

```
}
```

```
?>
```

```
</html>
```

Functions.php

<?php

```
function connectToDB() {
    global $link, $dbhost, $dbuser, $dbpass, $dbname;

    ($link = mysql_pconnect("$dbhost", "$dbuser", "$dbpass")) || die("Couldn't
connect to MySQL");

    // select db:
    mysql_select_db("$dbname", $link) || die("Couldn't open db: $dbname. Error if
any was: ".mysql_error() );
} // end func dbConnect();


function process_order_table($o,$item){

global $link;

$query="INSERT INTO orders (order_id,item) VALUES('$o','$item')";
$result=mysql_query($query, $link) or die("Died inserting login info into db.
Error returned if any: ".mysql_error());


}

function count_item_frequency(){
    global $link;

    $query="SELECT * FROM orders";
    $result1=mysql_query($query, $link) or die("Died inserting login info into db.
Error returned if any: ".mysql_error());
    while($row=mysql_fetch_array($result1)){

        $item_id=$row["item"];

        $query="SELECT item_id FROM item_freq WHERE item_id= '$item_id' ";
        $result2=mysql_query($query, $link) or die("Died inserting login info into db.
Error returned if any: ".mysql_error());
        $nums=mysql_num_rows ($result2);

        if ($nums == 0){

            $query="SELECT COUNT(item) FROM orders WHERE item ='$item_id' ";
            $result3=mysql_query($query, $link) or die("Died inserting login info into db.
Error returned if any: ".mysql_error());
            $row=mysql_fetch_array($result3);
            $frequency = $row["COUNT(item)"];
        }
    }
}
```



```

        $query="INSERT INTO item_freq (item_id,frequency)
VALUES('$item_id','$frequency') ";
        $result=mysql_query($query, $link) or die("Died inserting login info into db.
Error returned if any: ".mysql_error());
    }
}
    return true;
}

function printab(){
    echo "hello";
}

function genereate_tree(){
    global $link;

    $count1 = 1000;
    $var = 1;
    $count=1;
    while ($var <= $count1){

        $query= "SELECT Orders.order_id, orders.item, item_freq.frequency
FROM orders,item_freq WHERE orders.item = item_freq.item_id
AND orders.order_id = '$var'
ORDER BY frequency DESC";

        $result1=mysql_query($query, $link) or die("Died inserting login info into db.
Error returned if any: ".mysql_error());

        while($row=mysql_fetch_array($result1)){

            $parent_id = 0;

            $item_id = $row["item"];

            echo $item_id;
            $query="SELECT id,parentid,sup,conf FROM tree WHERE id = '$item_id' AND
parentid = '$parent_id' ";

            $result2=mysql_query($query, $link) or die("Died inserting login info into db.
Error returned if any: ".mysql_error());

            $nums=mysql_num_rows ($result2);

            echo $nums;
            echo "while er vitore";

```

```

if ($nums == 0){
    echo "if er vitore";

    $id = $item_id;

    $sup = 1.00/$count;
    $conf = 1.00/$count;

    $query="INSERT into TREE (id, count, sup, conf, parentid) values (
'$id', '$count', '$sup', '$conf','$parent_id')";

    $result3 = mysql_query($query, $link) or die("Died inserting login info into
db. Error returned if any: ".mysql_error());
    $parent_id = $item_id;
    }

else {

    echo "else er vitore";

    $temp = mysql_fetch_array($result2);

    $id = $temp["id"];
    $parent_id = $temp["parentid"];
    $sup = (($temp["sup"]* $count) + 1.00)/$count;
    $conf = 1.00/$count;

    $query = "UPDATE tree SET sup = '$sup' ,conf = '$conf' WHERE id = '$id' AND
parentid = '$parent_id'";
    $result3 = mysql_query($query, $link) or die("Died inserting login info into
db. Error returned if any: ".mysql_error());
    }

    $parent_id = $item_id;
    $count=$count+1;
    }
    $var = $var+1;
}

return true;

}
?>

```


Counter.java

```
import java.util.*;
```

```
public class Counter{
```

```
    static Orders order = new Orders();
```

```
    static Table item = new Table();
```

```
    static Table subcat = new Table();
```

```
    static Table cat = new Table();
```

```
    static Integer itemcount = 0;
```

```
    static Integer subcatcount = 0;
```

```
    static Integer catcount = 0;
```

```
    Sorter srt = new Sorter();
```

```
    public String toString(){
```

```
        return "" + order.toString() + "\n" + "\n" + "\n" + item.toString() + "\n" + " item count = " + itemcount +  
        "\n" + "\n" + "\n" + subcat.toString() + "\n" + " subcatcount = " + subcatcount + "\n" + "\n" + "\n"
```

```
        + cat.toString() + "\n" + " categorycount = " + catcount;
```

```
    }
```

```
    public void count(){
```

```
        ArrayList<Integer> tmp;
```

```
        for (int c = 1; c <= order.ordr.size(); c++){
```

```

tmp = order.get(c);
for(int i = 0; i < tmp.size(); i++){

    item.update(tmp.get(i)); itemcount++;
    subcat.update(tmp.get(i)/10); subcatcount++;
    cat.update(tmp.get(i)/100); catcount++;

}
}

}

public void sortItem(){

    int i = 1;

    while (i <= order.size()){

        ArrayList al = srt.sortItem(order.get(i), this.item);
        order.update(i, al);
        i++;
    }
}

```



```
}
```

```
public void sortSubCat(){
```

```
    int i = 1;
```

```
    while (i <= order.size()){
```

```
        ArrayList al = srt.sortSubCat(order.get(i), this.subcat);
```

```
        order.update(i, al);
```

```
        i++;
```

```
    }
```

```
}
```

```
public void sortCat(){
```

```
    int i = 1;
```

```
    while (i <= order.size()){
```

```
        ArrayList al = srt.sortCat(order.get(i), this.cat);
```

```
        order.update(i, al);
```

```
        i++;
```

```
    }
```

```
}}
```

FPTree.java

import java.util.ArrayList;

import java.util.Hashtable;

public class FPTree{

 Node rt;

 // Counter cnt = new Counter();

 // cnt.count();

 // cnt.sortItem();

 //

 Node generate(Orders Ord){

 Hashtable<Integer, ArrayList<Integer>> order = Ord.ordr;

 ArrayList lst;

 int c = order.size();

 rt = new Node(c);

 Node crnt;

 for(int i = 1; i <= order.size(); i++) {

 crnt = rt;

 lst = order.get(i);


```

Object[] tmp = lst.toArray();

for(int j = 0; j < tmp.length; j++) {
    //crnt = rt;

    Integer n = (Integer)tmp[j];

    int k = crnt.contains(n);

    if (k == -1){

        Node tp = new Node((int)n, (int)c, crnt);
        tp.parent = crnt;
        crnt = tp;
    } else {

        ArrayList<Node> st = crnt.chldrn;
        Object[] tp = st.toArray();
        Node tx = (Node)tp[k];
        tx.update();
        tp[k] = tx;

        ArrayList<Node> al = new ArrayList<Node>();

        for (int q = 0; q < tp.length; q++){

```

```

    al.add((Node)tp[q]);

}

crnt.chldrn = al;
crnt = tx;
}

}

//crnt = rt;

} return rt;
}

void prune(float sup, float conf){

}

}

```


Node.java

```
import java.util.ArrayList;
```

```
class Node {
```

```
    double sup, conf;
```

```
    int key, count, size;
```

```
    ArrayList<Node> chldrn = new ArrayList<Node>();
```

```
    Node parent;
```

```
    public Node(int k, int sz, Node nd){
```

```
        this.key = k;
```

```
        this.size = sz;
```

```
        count = 0;
```

```
        parent = nd;
```

```
        nd.chldrn.add(this);
```

```
        sup = ++count / size;
```

```
        conf = count/parent.count;
```

```
    }
```

```
    public Node(Integer k){
```

```
        this.key = k;
```

```
        //this.size = parent.size;
```

```
        count = 0;
```

```
        sup = ++count;
```

```

conf = count;
}

int contains(Integer n){

    if (chldrn.isEmpty()) {
        return -1;
    } else {
        Object[] tmp = chldrn.toArray();

        for (int i = 0; i<tmp.length; i++) {

            Node nd = (Node)tmp[i];
            if (nd.key == n){
                return i;
            }
        }
    }
    return -1;
}

```

```

public void update(){

```



```

++this.count;

this.sup = this.count/size;

this.conf = this.count/parent.sup;

}

void add(Node nd){

    chldrn.add(nd);
}

public boolean equals(Node nd){
    if ((this.parent != null) && (nd.parent != null)){
        return ( this.key == nd.key && this.parent.equals(nd.parent));
    }else {
        return this.key == nd.key;
    }
}

public String toString(){

```

```

if (this.parent == null && !this.chldrn.isEmpty()){
String tmp = " current node is root the children are "+ this.chldrn.toString();
return tmp;
}else if(this.chldrn.isEmpty()){
String tmp = " key of the node is " + this.key;
return tmp;
}
else{
String tmp = " key of the node is " + this.key + " the children are "+ this.chldrn.toString()+" and the
parent is "+ this.parent.toString();
return tmp;
}
}
}
}

```


Orders.java

```
import java.util.Hashtable;
```

```
import java.util.ArrayList;
```

```
public class Orders{
```

```
    static Hashtable<Integer, ArrayList<Integer>> order = new Hashtable<Integer, ArrayList<Integer>>();
```

```
    public Orders(){
```

```
        System.out.println("generating orders");
```

```
        generate();
```

```
    }
```

```
    public void generate(){
```

```
        Integer n, val;
```

```
        double i, j, k;
```

```
        n = 1;
```

```
        while(n <= 10){
```

```
            ArrayList<Integer> lst = new ArrayList<Integer>();
```

```
            while (lst.size() < 5){
```

```
                i = Math.random()*10;
```

```
                j = Math.random()*10;
```

```

    k = Math.random()*10;

    if ((i >= 1 )&& (j >= 1) && (k >= 1)){
        val = ((int)i) + ((int)j)*10 + ((int)k)*100;
        if (lst.contains(val) == false ){
            lst.add(val);
        }
    }
}

ordr.put(n, lst);

n++;
}
}

public String toString(){

    //long time = System.currentTimeMillis();

    //System.out.println("toString() on Orders invoked at " + time + " milliseconds");

    String tmp = ordr.toString();

    //System.out.println(" method terminating at " + (System.currentTimeMillis() - time) + "
milliseconds");

    return tmp;
}

public int size(){
    return ordr.size();
}

```



```

public ArrayList<Integer> get(Integer i){
    //long time = System.currentTimeMillis();

    //System.out.println("get() on Orders invoked at " + time + " milliseconds");

    ArrayList<Integer> tmp = ordr.get(i);

    //System.out.println(" method terminating at " + (System.currentTimeMillis() - time) + "
milliseconds");

    return tmp;
}

ArrayList<Integer> update(Integer key, ArrayList<Integer>val){

    //long time = System.currentTimeMillis();

    //System.out.println("updating entry "+ key+ " " + "with value "+ val +" in items starting at " + time + "
milliseconds");

    ordr.remove(key);

    ArrayList<Integer> tmp = ordr.put(key, val);

    //System.out.println(" method terminating at " + (System.currentTimeMillis() - time) + "
milliseconds");

    return tmp;
}

public void remove(Integer key){
    ordr.remove(key);
}
}

```

Sorter.java

```
import java.util.*;
```